Lab Manual

for

Linear Algebra

by

Jim Hefferon

Cover: my Chocolate Lab, Suzy.

Contents

Python and Sage 1
Gauss's Method 13
Vector Spaces 25
Matrices 33
Maps 43

Matrices

Matrix operations are mechanical, and are therefore perfectly suited for mechanizing.

Defining

To define a matrix you can use real number entries, or complex entries, or entries from other number systems such as the rationals.

```
sage: A = matrix(RDF, [[1, 2], [3, 4]])
sage: A
[1.0 2.0]
[3.0 4.0]
sage: i = CDF(i)
sage: A = matrix(CDF, [[1+2*i, 3+4*i], [5+6*i, 7+8*i]])
sage: A
[1.0 + 2.0*I 3.0 + 4.0*I]
[5.0 + 6.0*I 7.0 + 8.0*I]
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: A
[1 2]
[3 4]
```

As with the rea numbers, Sage has two choices for models of the complex numbers and we use here the one connected with the computer's hardware. By default Sage uses i for the square root of -1 (in contrast with Python, which uses j). Note that before working with complex numbers we reset i because that letter is used for many things, so resetting is a good habit.

Unless we have a reason to do otherwise, in this chapter we'll use rational numbers because the matrices easier to read — 1 is easier than 1.0 — and because the matrices in the book have rational entries.

The `matrix` constructor allows you to specify the number of rows and columns.

```
sage: B = matrix(QQ, 2, 3, [[1, 1, 1], [2, 2, 2]])
sage: B
[1 1 1]
[2 2 2]
```

If your specified size doesn't match the entries

```
sage:
```

then Sage's error says `Number of rows does not match up with specified number`. Until now we've let Sage figure out the matrix's number of rows and columns size from the entries but a shortcut to get the zero matrix is to put the number zero in the place of the entries, and there you must say which size you want.

```
sage: B = matrix(QQ, 2, 3, 0)
sage: B
[0 0 0]
[0 0 0]
```

Another place where specifying the size is a convienence is Sage's shortcut to get an identity matrix.

```
sage: B = matrix(QQ, 2, 2, 1)
sage: B
[1 0]
[0 1]
```

The difference between this shortcut and the prior one is that `matrix(QQ, 3, 2, 1)` gives an error because an identity matrix must be square. Sage has another shortcut that can't lead to this error.

```
sage: I = identity_matrix(4)
sage: I
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

Sage has a wealth of methods on matrices. For instance, you can transpose the rows to columns or test if the matrix is *symmetric*, unchanged by transposition.

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: A.transpose()
[1 3]
[2 4]
sage: A.is_symmetric()
False
```

Linear combinations

Addition and subtraction are natural.

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: B = matrix(QQ, [[1, 1], [2, -2]])
sage: A+B
[2 3]
[5 2]
sage: A-B
[0 1]
[1 6]
sage: B-A
[ 0 -1]
[-1 -6]
```

Sage knows that adding matrices with different sizes is undefined; this gives an error.

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: C = matrix(QQ, [[0, 0, 2], [3, 2, 1]])
sage: A+C
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call

/home/ftpmaint/linear-algebra/lab/<ipython console> in <module>()

/usr/local/src/sage-5.2-linux-32bit-ubuntu_12.04_lts-i686-Linux/local/

/usr/local/src/sage-5.2-linux-32bit-ubuntu_12.04_lts-i686-Linux/local/

TypeError: unsupported operand parent(s) for '+': 'Full MatrixSpace of
           2 by 2 dense matrices over Rational Field' and 'Full
           MatrixSpace of 2 by 3 dense matrices over Rational Field'
```

Some of the lins in that output block don't contain key information so the've just been truncated. But the last line (which is so long it had to be broken and wrapped twice) is where the action is. It says that the `+` operand is not defined between a 2×2 matrix and a 2×3 matrix.

Scalar multiplication is also natural so you have linear combinations.

```
sage: 3*A
[ 3  6]
[ 9 12]
sage: 3*A-4*B
[-1  2]
[ 1 20]
```

Multiplication

Matrix-vector product Matrix-vector multiplication is just what you would guess.

```
sage: A = matrix(QQ, [[1, 3, 5, 9], [0, 2, 4, 6]])
sage: v = vector(QQ, [1, 2, 3, 4])
sage: A*v
(58, 40)
```

The 2×4 matrix A multiplies the 4×1 column vector $\vec{v}$, with the vector on the right side, as $A\vec{v}$.

If you try this vector on the left as `v*A` then Sage gives a mismatched-sizes error.

```
sage: A = matrix(QQ, [[1, 3, 5, 9], [0, 2, 4, 6]])
sage: v = vector(QQ, [1, 2, 3, 4])
sage: v*A
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call

/home/ftpmaint/linear-algebra/lab/<ipython console> in <module>()

/usr/local/src/sage-5.2-linux-32bit-ubuntu_12.04_lts-i686-Linux/local/

/usr/local/src/sage-5.2-linux-32bit-ubuntu_12.04_lts-i686-Linux/local/
```

```

TypeError: unsupported operand parent(s) for '*': 'Vector space of
                dimension 4 over Rational Field' and 'Full MatrixSpace
                of 2 by 4 dense matrices over Rational Field'
```

As in the earlier error traceback some lines are truncated and the final line is wrapped twice to show it all. That final line says, in short, that the product of a size 4 vector with a size 2×4 matrix is not defined.

Of course you can multiply from the left by a vector if it has a size that matches the matrix.

```
sage: w = vector(QQ, [3, 5])
sage: w*A
(3, 19, 35, 57)
```

In practice you will see matrix-vector multiplications done with vectors on the left, and others on the right. In the book we multiply on the right, just to settle on one rather than switch around arbitrarily (and to have the matrix application $H\vec{x}$ fit with the map application $h(x)$). Sage will do either, although in some ways it has a preference for the vector on the left (as we will see in Chapter 5).

Matrix-matrix product If the sizes match then Sage will multiply the matrices. Here is the product of a 2×2 matrix A and a 2×3 matrix B.

```
sage: A = matrix(QQ, [[2, 1], [4, 3]])
sage: B = matrix(QQ, [[5, 6, 7], [8, 9, 10]])
sage: A*B
[18 21 24]
[44 51 58]
```

Trying `B*A` gives an error starting `TypeError: unsupportedoperand parent(s) for '*'` , meaning that the product operation in this order is undefined.

Same-sized square matrices have the product defined in either order.

```
sage: A = matrix(QQ, [[1, 2], [3, 4]])
sage: B = matrix(QQ, [[4, 5], [6, 7]])
sage: A*B
[16 19]
[36 43]
sage: B*A
[19 28]
[27 40]
```

They are different; matrix multiplication is not commutative.

```
sage: A*B == B*A
False
```

In fact, matrix multiplication is very noncommutative in that if you produce two $n\times n$ matrices at random then they almost surely don't commute. Sage lets us produce matrices at random.

```
sage: random_matrix(RDF, 3, min=-1, max=1)
[  0.839578752373   -0.513434560745  -0.0103952682343]
[ -0.650255834117   0.0742614255301   -0.535124891715]
[  0.244339416814   -0.875227855986   -0.272810898333]
sage: random_matrix(RDF, 3, min=-1, max=1)
```

```
[  -0.466031089759    0.853311532228    -0.65308044199]
[   0.347679614686   -0.374855889227    -0.83221775243]
[  -0.286374949422  -0.0784330500423   -0.470870342834]
```

(Note the `RDF`. We prefer real number entries here because `random_matrix` is more straightforward in this case than in the rational entry case.)

```
sage: number_commuting = 0
sage: for n in range(1000):
....:     A = random_matrix(RDF, 2, min=-1, max=1)
....:     B = random_matrix(RDF, 2, min=-1, max=1)
....:     if (A*B == B*A):
....:         number_commuting = number_commuting + 1
....:
sage: print "number commuting of 1000=",number_commuting
number commuting of 1000= 0
```

Inverse Recall that if A is nonsingular then its *inverse* A^{-1} is the matrix such that $AA^{-1} = A^{-1}A$ is the identity matrix.

```
sage: A = matrix(QQ, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
sage: A.is_singular()
False
```

We have a formula for 2×2 matrix inverses but in the book to by-hand compute inverses for larger matrices we write the original matrix next to the identity, and then do Gauss-Jordan reduction.

```
sage: I = identity_matrix(3)
sage: B = A.augment(I, subdivide=True)
sage: B
[ 1  3  1| 1  0  0]
[ 2  1  0| 0  1  0]
[ 4 -1  0| 0  0  1]
sage: C = B.rref()
sage: C
[    1     0     0|    0   1/6   1/6]
[    0     1     0|    0   2/3  -1/3]
[    0     0     1|    1 -13/6   5/6]
```

The inverse is the resulting matrix on the right.

```
sage: A_inv = C.matrix_from_columns([3, 4, 5])
sage: A_inv
[    0   1/6   1/6]
[    0   2/3  -1/3]
[    1 -13/6   5/6]
sage: A*A_inv
[1 0 0]
[0 1 0]
[0 0 1]
sage: A_inv*A
[1 0 0]
[0 1 0]
[0 0 1]
```

Since this is an operation that Sage users do all the time, there is a standalone command.

```
sage: A_inv = A.inverse()
sage: A_inv
[    0   1/6   1/6]
[    0   2/3  -1/3]
[    1 -13/6   5/6]
```

One reason for finding the inverse is to make solving linear systems easier. These three systems

$$\begin{array}{rl} x+3y+z &= 4 \\ 2x+y &= 4 \\ 4x-y &= 4 \end{array} \qquad \begin{array}{rl} x+3y+z &= 2 \\ 2x+y &= -1 \\ 4x-y &= 5 \end{array} \qquad \begin{array}{rl} x+3y+z &= 1/2 \\ 2x+y &= 0 \\ 4x-y &= 12 \end{array}$$

share the matrix of coefficents but have different vectors on the right side. If you have first calculated the inverse of the matrix of coefficients then solving each system takes just a matrix-vector product.

```
sage: A = matrix(QQ, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
sage: A_inv = A.inverse()
sage: v1 = vector(QQ, [4, 4, 4])
sage: v2 = vector(QQ, [2, -1, 5])
sage: v3 = vector(QQ, [1/2, 0, 12])
sage: A_inv*v1
(4/3, 4/3, -4/3)
sage: A_inv*v2
(2/3, -7/3, 25/3)
sage: A_inv*v3
(2, -4, 21/2)
```

Running time

Since computers are fast and accurate they open up the possibility of solving problems that are quite large. Large linear algebra problems occur frequently in science and engineering. In this section we will suggest what limits there are to computers. (In this section we will use matrices with real entries because they are common in applications.)

One of the limits on just how large a problem we can do is how quickly the computer program can give answers. Naturally computers take longer to perform operations on matrices that are larger but it may be that the time the program takes to compute the answer grows more quickly than does the size of the problem — for instance, when the size of the matrix doubles then the time to do the job more than doubles.

We will time the matrix inverse operation. This is an important operation; for instance, if we could do large matrix inverses quickly then we could quickly solve large linear systems, with just a matrix-vector product. We also use it because it makes a good illustration.

```
sage: A = matrix(RDF, [[1, 3, 1], [2, 1, 0], [4, -1, 0]])
sage: A
[ 1.0  3.0  1.0]
[ 2.0  1.0  0.0]
[ 4.0 -1.0  0.0]
```

```
sage: A.is_singular()
False
sage: timeit('A.inverse()')
5 loops, best of 3: 208 µs per loop
```

Note that Sage's `timeit` runs the command many times and makes a best guess about how long the operation ideally takes, because on any one time your computer may have been slowed down by a disk write or some other interruption.

The inverse operation took on the order of hundreds of microseconds. A single microsecond is 0.000 001 seconds. That's fast, but then A is only a 3×3 matrix.

Worse, A is a particular 3×3 matrix, and you'd like to know how long it takes to invert a generic, or average, matrix. You could try finding the inverse of a random matrix.

```
sage: timeit('random_matrix(RDF, 3, min=-1, max=1).inverse()')
625 loops, best of 3: 124 µs per loop
sage: timeit('random_matrix(RDF, 3, min=-1, max=1).inverse()')
625 loops, best of 3: 123 µs per loop
sage: timeit('random_matrix(RDF, 3, min=-1, max=1).inverse()')
625 loops, best of 3: 124 µs per loop
```

Again this is of the order of hundreds of microseconds.

But this has the issue that we can't tell from it whether the time is spent generating the random matrix or finding the inverse. In addition, there is a subtler point: we also can't tell right away if this command generates many random matrices and finds each's inverse, or if it generates one random matrix and applies the inverse many times. This code at least makes that point clear. For various sizes, 3×3, 10×10, etc., if finds a single random matrix and then gets the time to compute its inverse.

```
sage: for size in [3, 10, 25, 50, 75, 100, 150, 200]:
....:     print "size=",size
....:     M = random_matrix(RR, size, min=-1, max=1)
....:     timeit('M.inverse()')
....:
size= 3
625 loops, best of 3: 125 µs per loop
size= 10
625 loops, best of 3: 940 µs per loop
size= 25
25 loops, best of 3: 12 ms per loop
size= 50
5 loops, best of 3: 92.4 ms per loop
size= 75
5 loops, best of 3: 308 ms per loop
size= 100
5 loops, best of 3: 727 ms per loop
size= 150
5 loops, best of 3: 2.45 s per loop
size= 200
5 loops, best of 3: 5.78 s per loop
```

Some of those times are in microseconds, some are in milliseconds, and some are in seconds. This table is consistently in seconds.

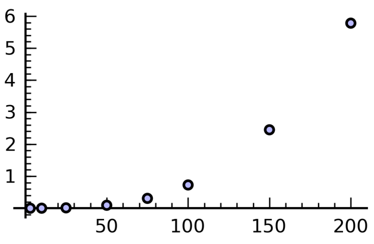
size	*seconds*
3	0.000 125
10	0.000 940
25	0.012
50	0.092 4
75	0.308
100	0.727
150	2.45
200	5.78

The time grows faster than the size. For instance, in going from size 25 to size 50 the time more than doubles: 0.0924/0.012 is 7.7. Similarly, increasing the size from 50 to 200 causes the time to increase by much more than a factor of four: $5.78/0.0924 \approx 62.55$.

To get a picture give Sage the data as a list of pairs.

```
sage: d = [(3, 0.000125), (10, 0.000940), (25, 0.012),
....:      (50, 0.0924), (75, 0.308), (100, 0.727),
....:      (150, 2.45), (200, 5.78)]
sage: g = scatter_plot(d)
sage: g.save("graphics/mat001.png")
```

(If you enter `scatter_plot(d)` at the prompt, that is, without saving it as g, then Sage will pop up a window with the graphic.)[1]



The graph dramatizes that the the ratio time/size is not constant since the data clearly does not lie on a line.

Here is some more data. The times are big enough that this computer had to run overnight.

```
sage: for size in [500, 750, 1000]:
....:         print "size=",size
....:     M = random_matrix(RR, size, min=-1, max=1)
....:     timeit('M.inverse()')
....:
size= 500
5 loops, best of 3: 89.2 s per loop
size= 750
5 loops, best of 3: 299 s per loop
size= 1000
5 loops, best of 3: 705 s per loop
```

[1]The graphics in this book were generated using a few more of Sage's drawing options than are shown here., both to make them look better and to better fit the page. For instance, this graphic was generated with `g = scatter_plot(d, markersize=10, facecolor='#b9b9ff')`. The Sage command used to get the plot here was `g.save("graphics/mat001.png", figsize=[2.25,1.5], axes_pad=0.05, fontsize=7, dpi=500)`. We shall omit much of this code about decoration as clutter. See the Sage manual for `plot` options.

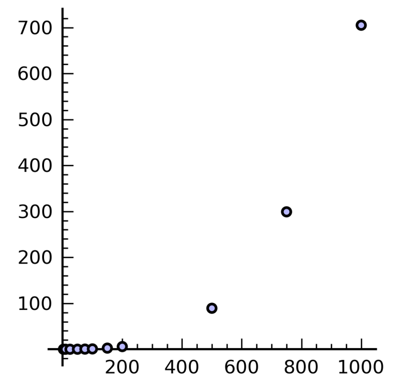
Again the table is a neater way to present the data.

size	*seconds*
500	89.2
750	299.
1000	705.

Get a graph by tacking the new data onto the existing data.

```
sage: d = d + [(500, 89.2), (750, 299), (1000, 705)]
sage: g = scatter_plot(d)
sage: g.save("graphics/mat002.png")
```

The result is this graphic.



Note that the two graphs have different scales; if you generated this graph with the same vertical scale as the prior graph then the data would go far off the top of the page.

So a practical limit to the size of a problem that we can solve with this matrix inverse operation comes from the fact that the graph above is not a line. The time required grows much faster than the size, and just gets too large.

A major effort in Computer Science is to find fast algorithms to do various tasks. Many people have worked on tasks in Linear Algebra in particular, such as finding the inverse of a matrix, because they are so common in applications.

Bibliography

Robert A. Beezer. Sage for Linear Algebra. http://linear.ups.edu/download/fcla-2.22-sage-4.7.1-preview.pdf, 2011.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

Jim Hefferon. Linear Algebra. http://joshua.smcvt.edu/linearalgebra, 2012.

David Joyner and William Stein. Open source mathematical software. *Notices of the AMS*, page 1279, November 2007.

Python Team. Floating point arithmetic: issues and limitations, 2012a. URL http://docs.python.org/2/tutorial/floatingpoint.html. [Online; accessed 17-Dec-2012].

Python Team. The python tutorial, 2012b. URL http://docs.python.org/2/tutorial/index.html. [Online; accessed 17-Dec-2012].

Ron Brandt. *Powerful Learning*. Association for Supervision and Curriculum Development, 1998.

Sage Development Team. Sage tutorial 5.3. http://www.sagemath.org/pdf/SageTutorial.pdf, 2012a.

Sage Development Team. Sage reference manual 5.3. http://www.sagemath.org/pdf/reference.pdf, 2012b.

Shunryu Suzuki. *Zen Mind, Beginners Mind*. Shambhala, 2006.